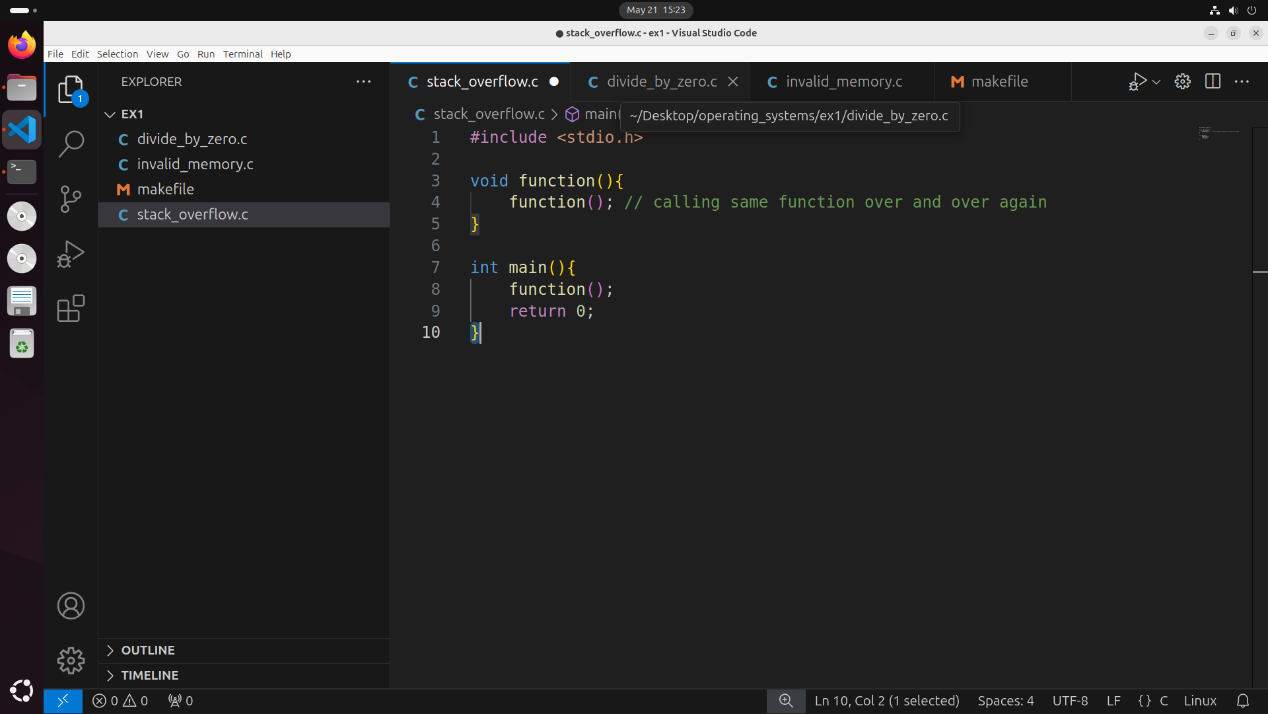
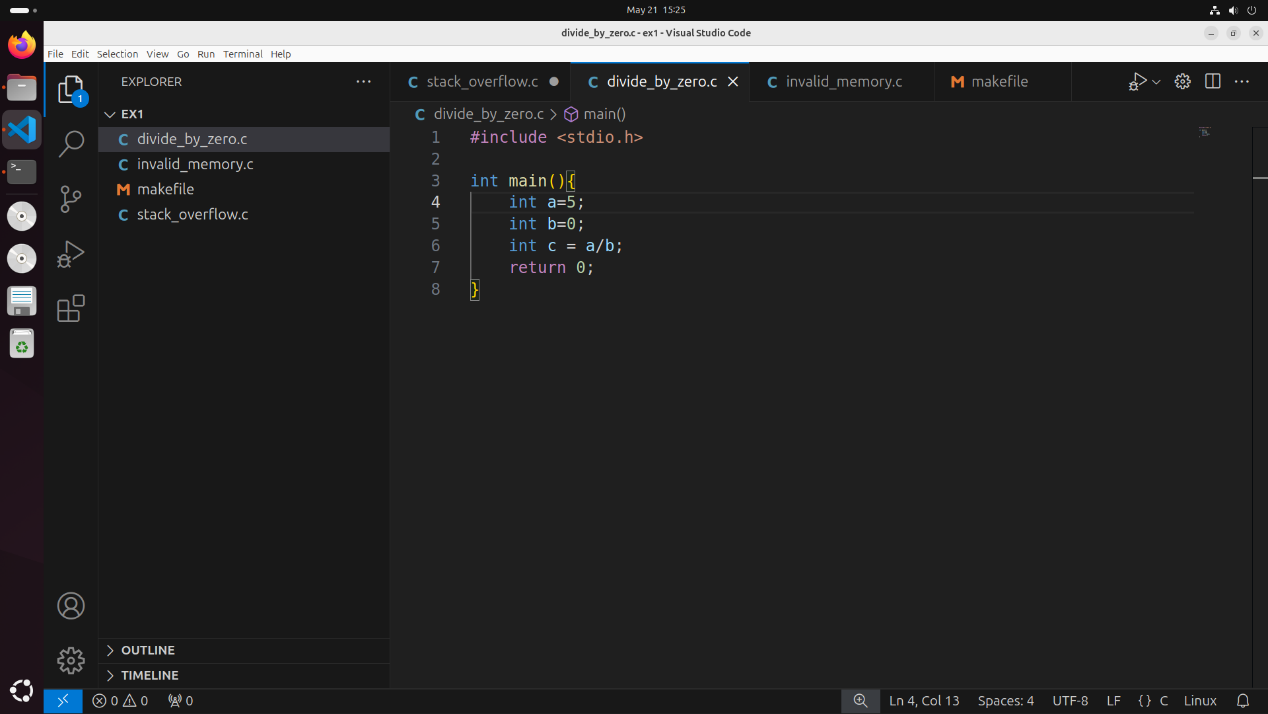
**Assignment 1 operating systems**

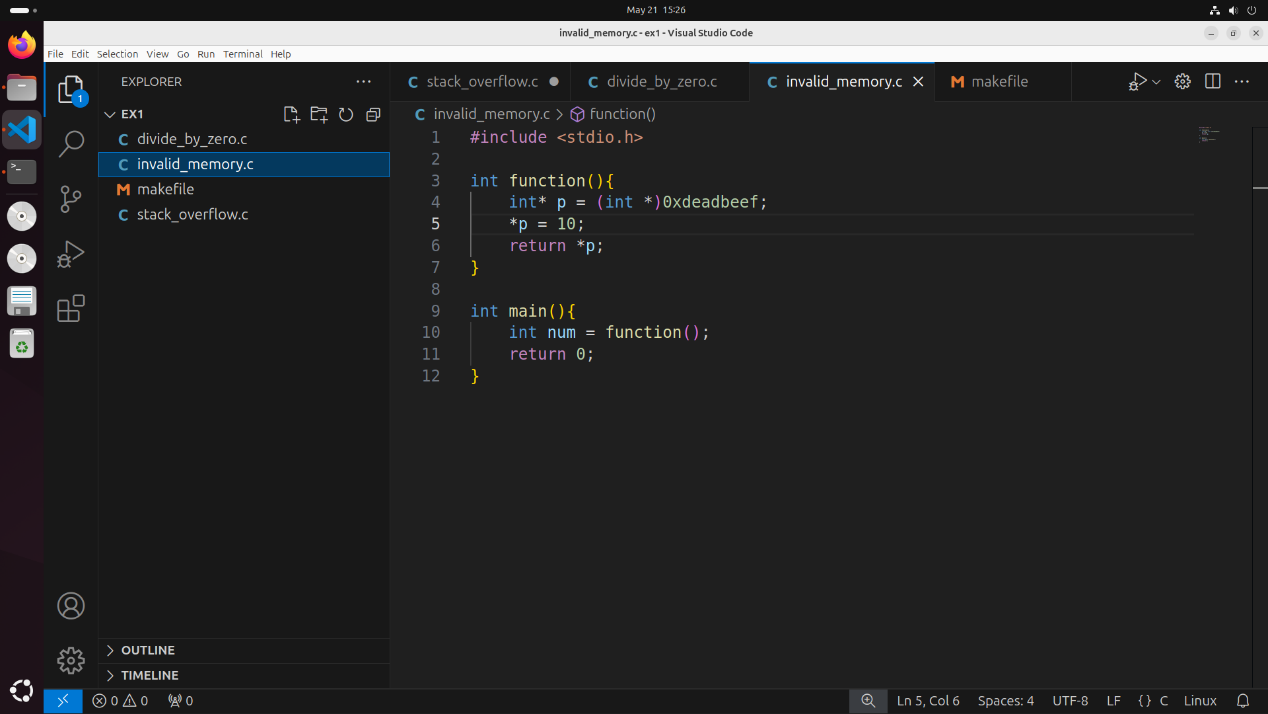
In this section, I wrote three programs that aim to crash the program in different ways: overflowing the stack, dividing by zero, and using undefined memory.

overflowing the stack: 

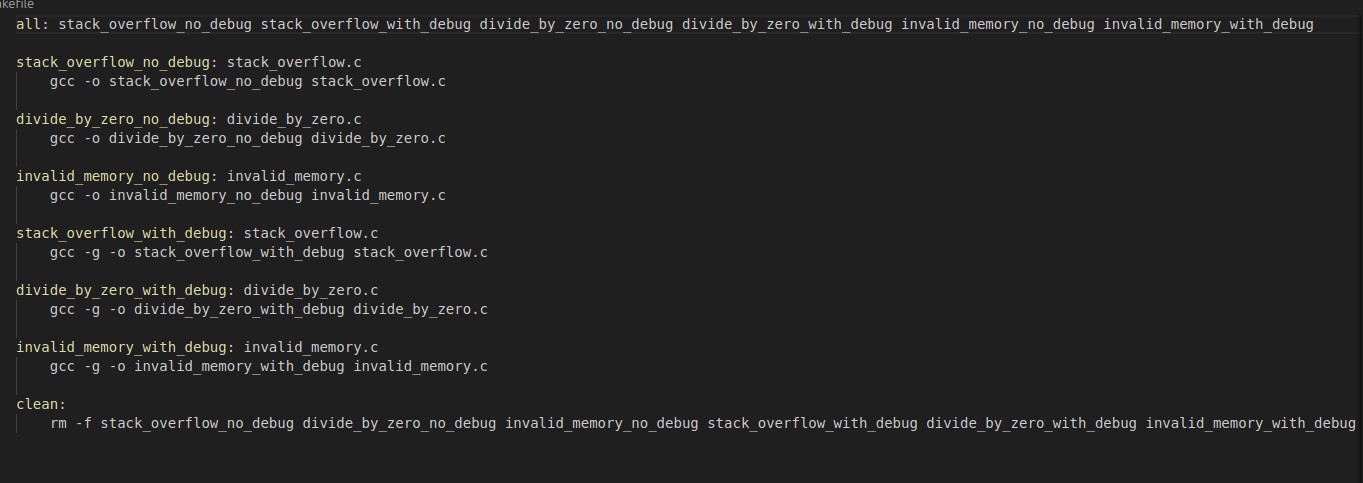
dividing by zero:



using undefined memory:



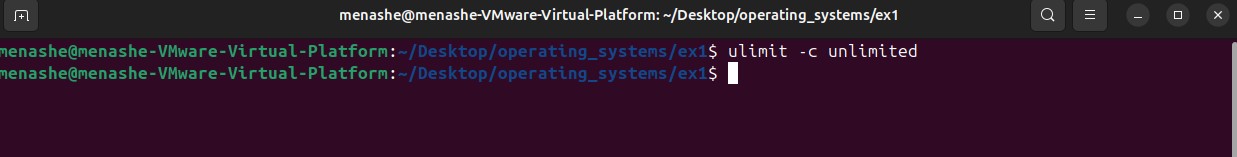
In addition, I wrote a Makefile for all the files, creating executable files with and without debug info:



After running the Makefile, two new files will be created for each C file (one with and one without debug info).

We will see all the following steps using screenshots - I intend to create, open, and analyze a core dump resulting from running a few simple programs in the C language. The process includes compiling the code with and without debug info, using a textual debugger (gdb), and a graphical debugger (ddd) to locate the crash and examine the variable values. Here is a detailed description of the steps:

First, let's ensure that the ulimit setting allows for the creation of core dumps:

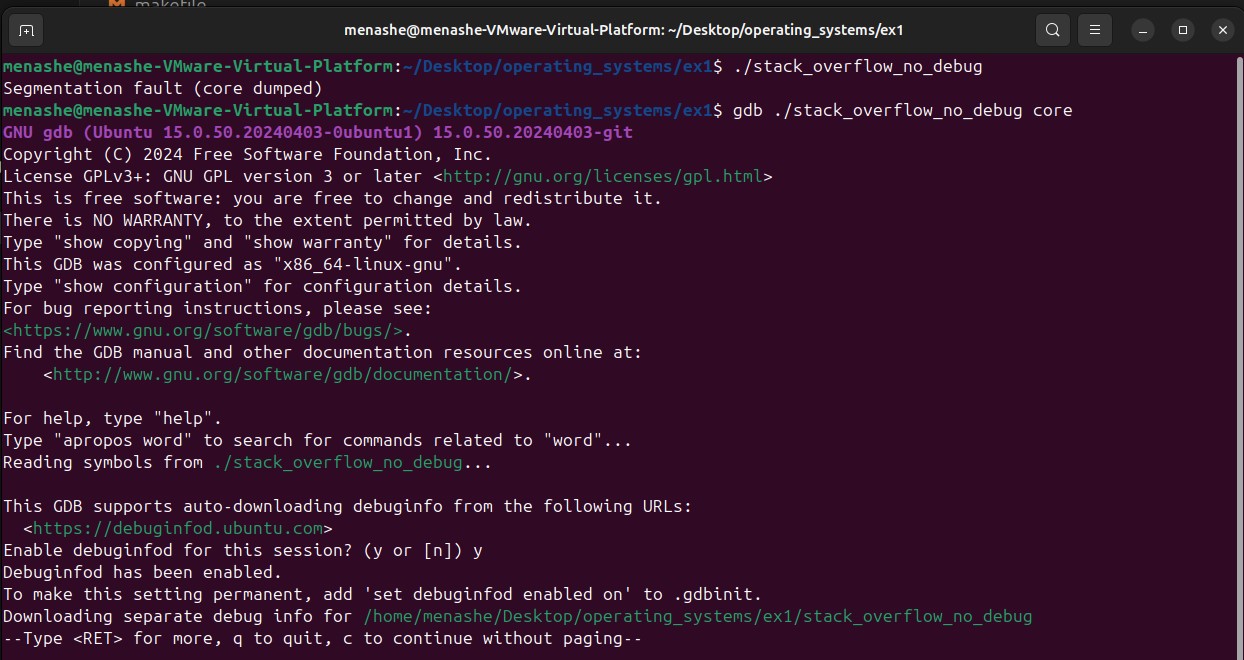


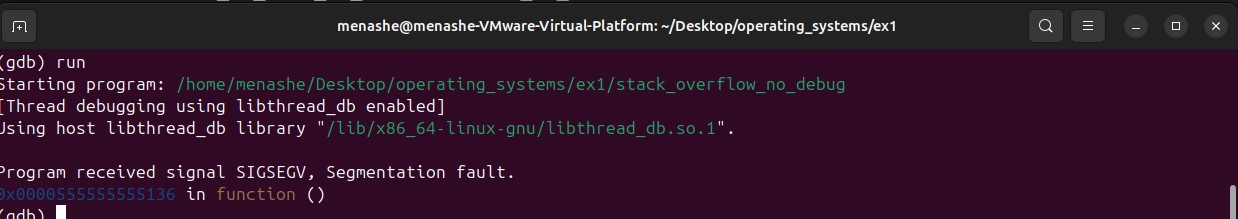
In the following demonstrations of the program behavior I will show the difference between having a debug info or not.

**first program:**

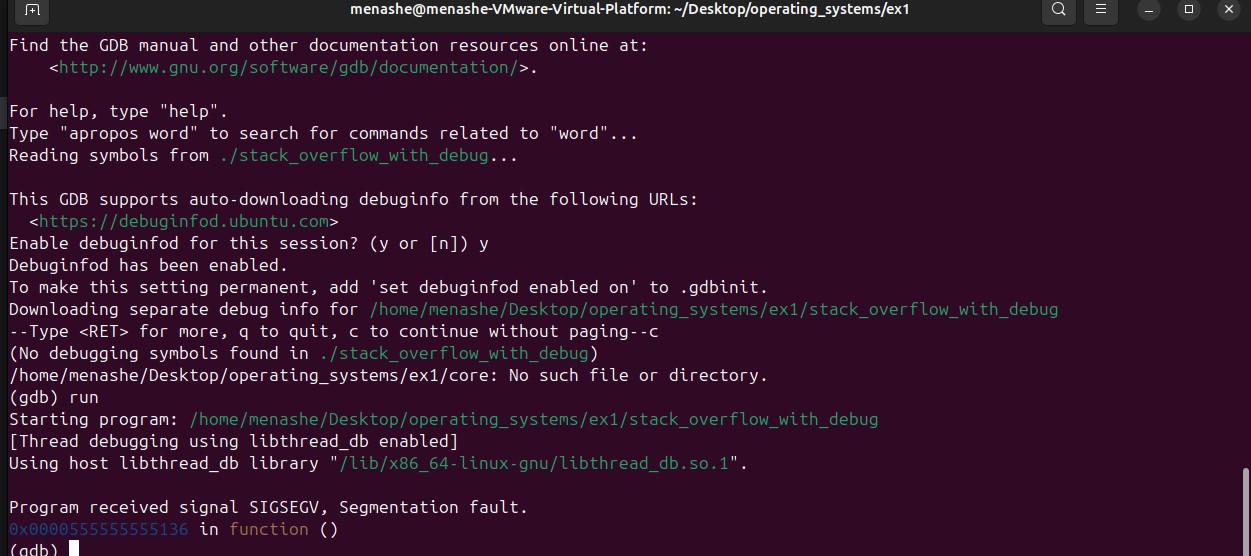
we will open the core file using gdb and run the program using the run command.

Without debug info:





With debug info:

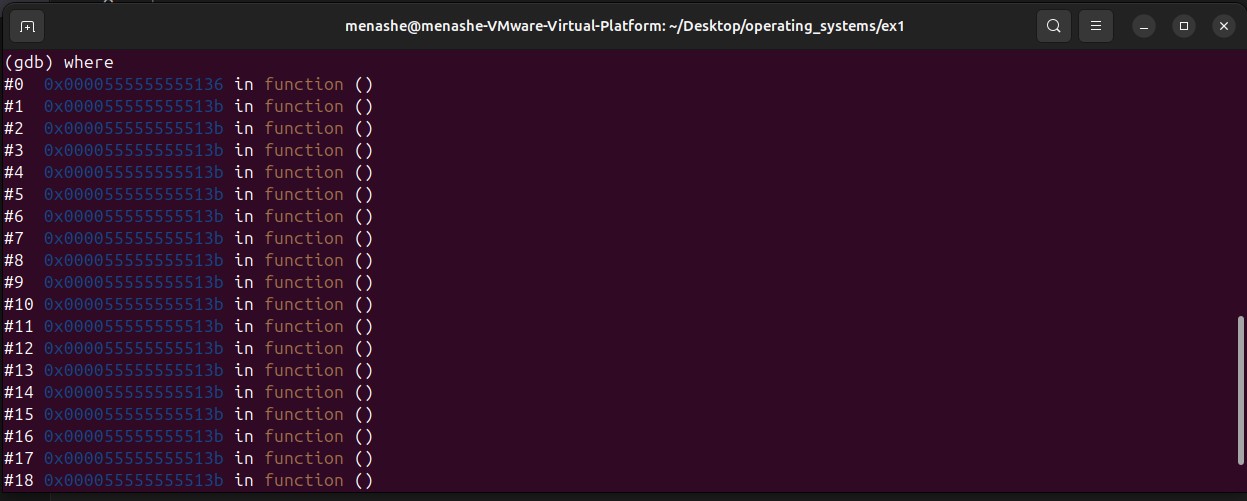


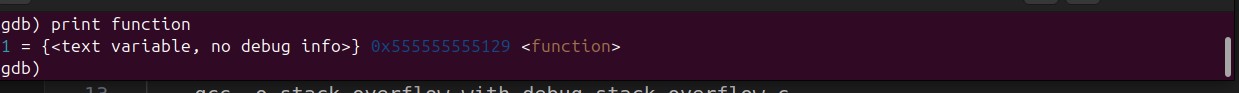


Notice the difference between them. With the debug info we got also file where there is error and exactly the number of line where the error occur.

In addition, we will run the where command to display the stack trace and see all the function calls leading up to the crash and We use the print command to print the value of a variable at the current breakpoint in the code. This is very useful in the debugging process when you want to check the value of a variable during program execution:

Without debug info:

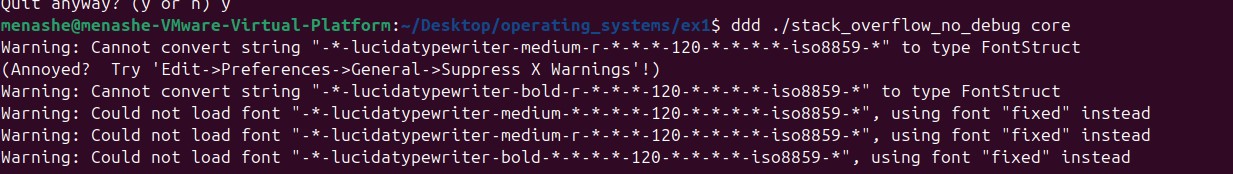




With debug info:

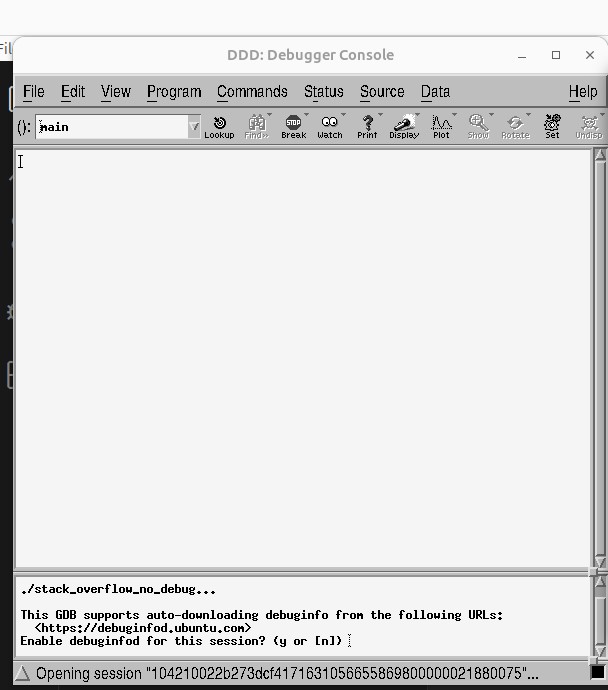




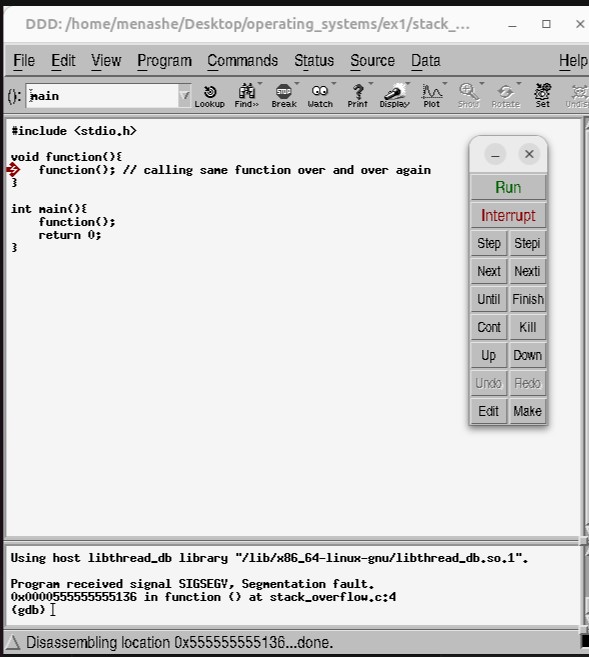


Now, we will debug using a graphical debugger, which allows for visual data representation and easier navigation through the code compared to a textual debugger alone. Note that since we created a file without debugging information, we won't be able to see the original code, variable names, or get detailed error information:

We will use the command – ddd ./Name\_Of\_The\_File core



With debug info:

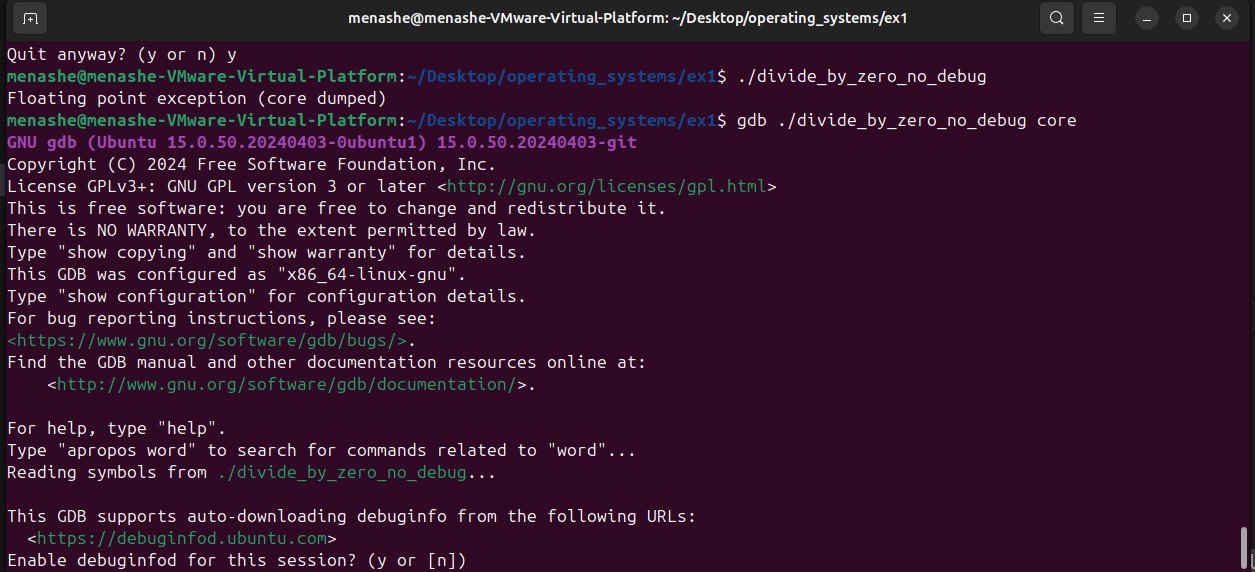


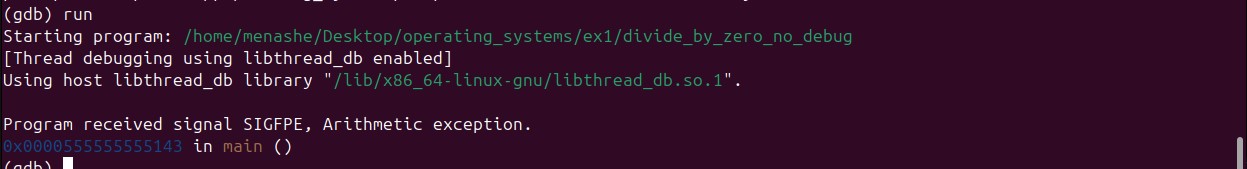
The debugging steps for the other programs are very similar. Here are the steps for programs 2 and 3:

**second program:**

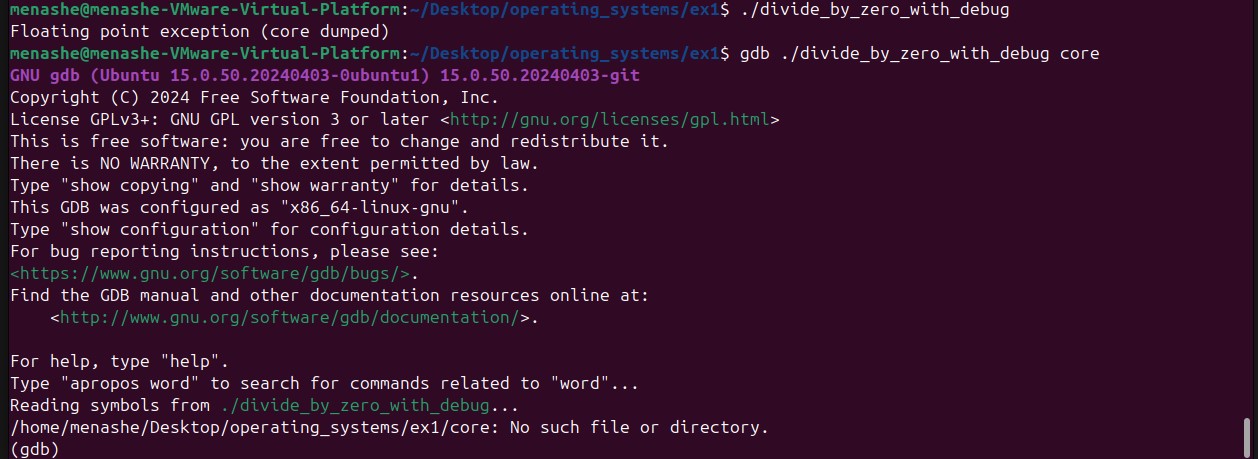
we will open the core file using gdb and run the program using the run command.

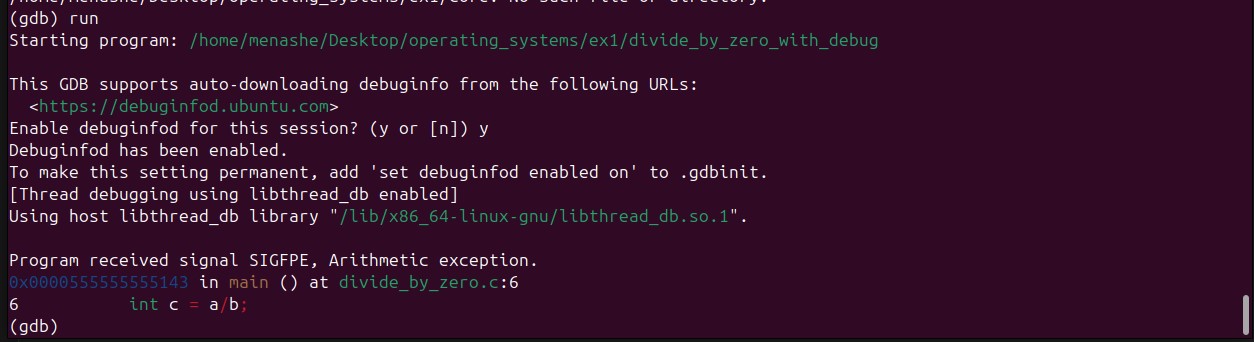
Without debug info:





with debug info:





We will run the where and print commands:

Without debug:



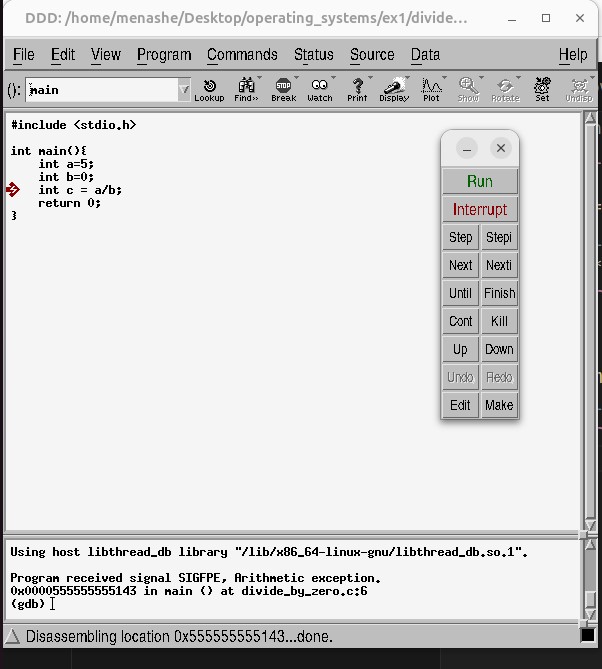
notice that since we are in a state without debug information, we will not be able to know the values of the variables during the code execution:

with debug:



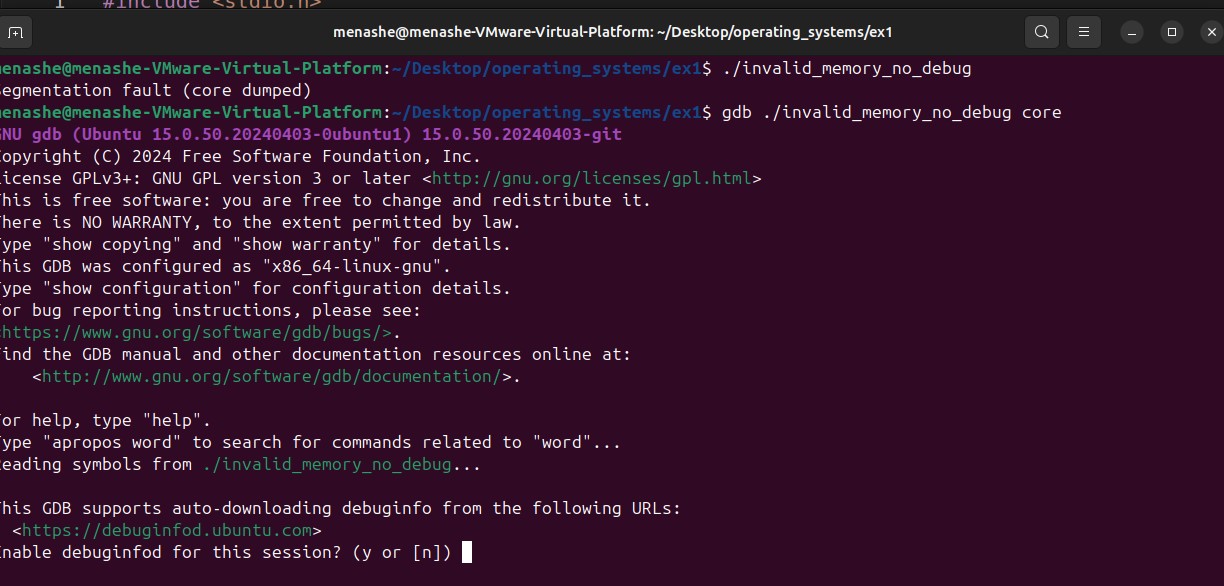
Here unlike the previous situation, here we can see the state of the variables at the time the program crashed and also identify which file caused the crash and on which line. It's easy to understand that the problem is with the variable **c**, which is being divided by zero.

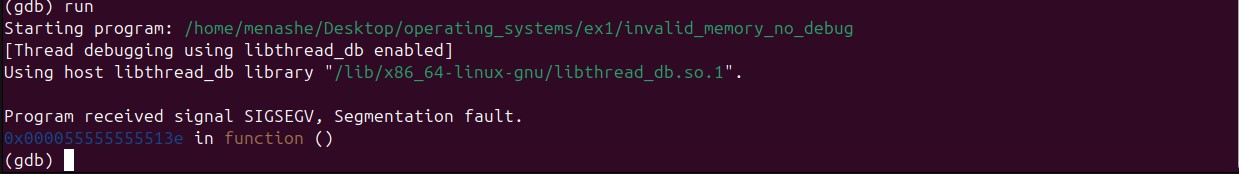
The graphical debugger without debug info will show a blank screen, so we will only show the graphical debugger with debug info.



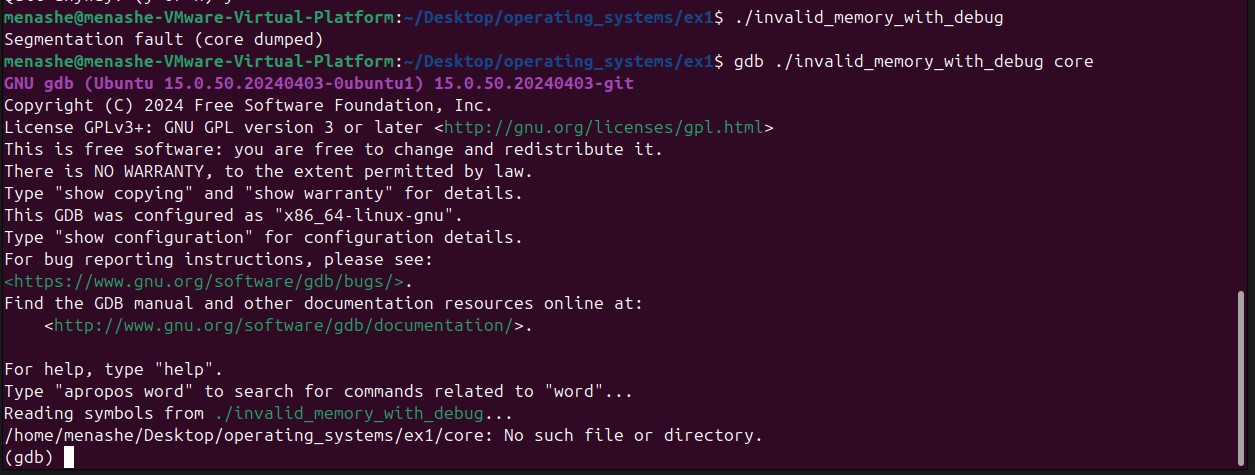
**third program:**

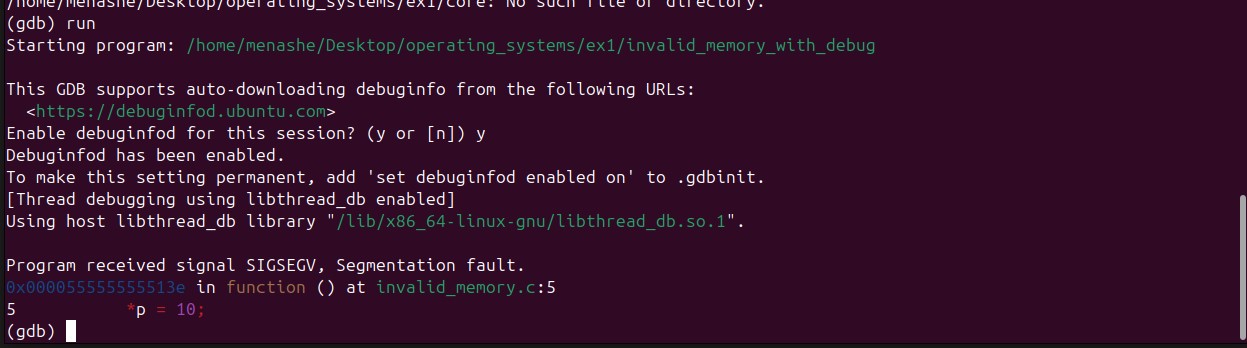
without debug info:





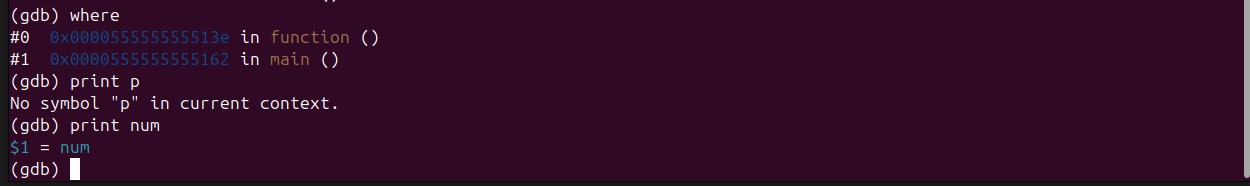
With debug info:



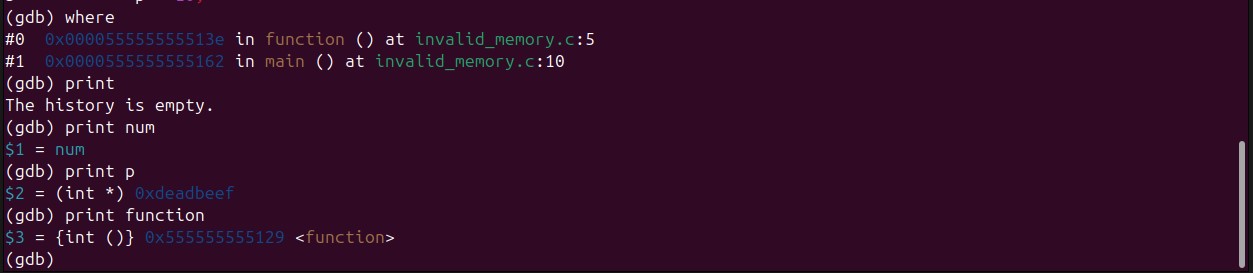


We will run the where and print commands:

Without debug info:

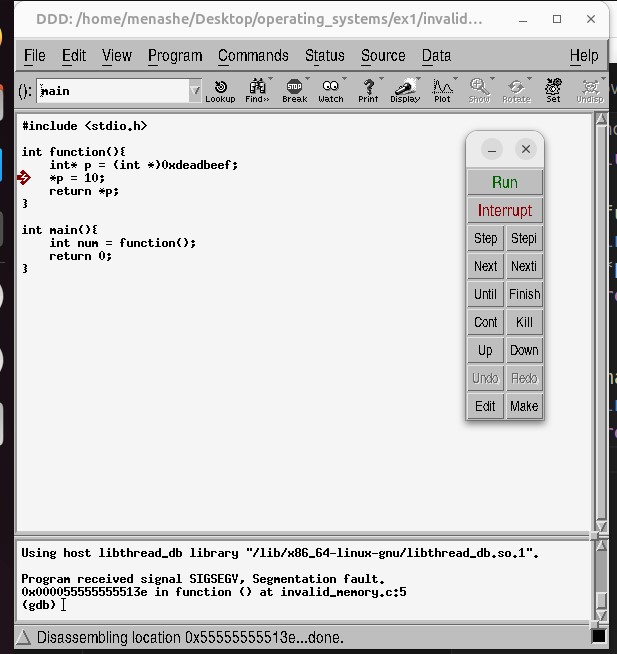


With debug info:

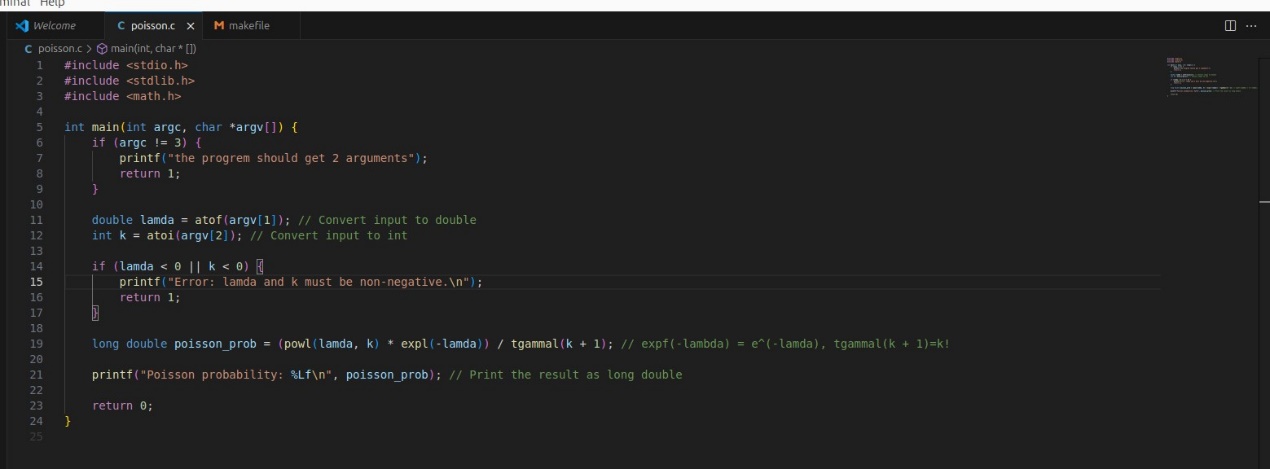


From here, it is easily noticeable that the error comes from the variable **p**, which we haven't allocated an appropriate memory space for.

Again, as before we will show the graphical debugger only for the program with the debugging option:



**Exercise 2:**

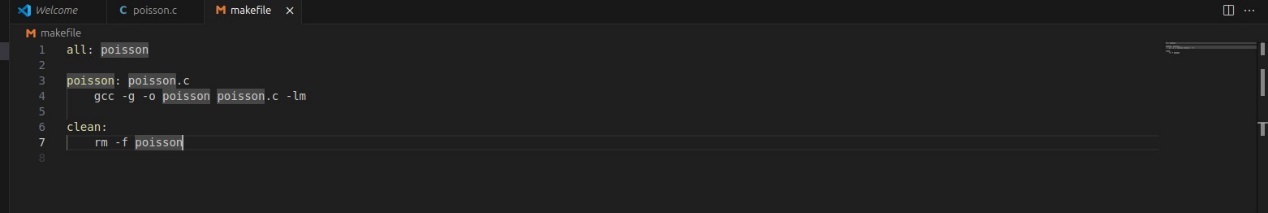


The program checks if exactly two arguments are provided. If not, it prints an error message and exits, It converts the input arguments to a floating-point number (**lamda**) and an integer (**k**), checks if the values of **lamda** and **k** are non-negative. If not, it prints an error message and exits.

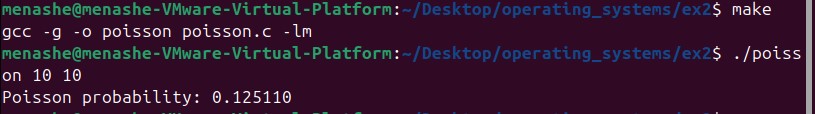
The program calculates the Poisson probability using the formula. The functions used are:

* powl to calculate 𝜆^k
* expl to calculate 𝑒^−𝜆.
* tgammal to calculate (𝑘+1)!, which is the factorial of 𝑘*!*.

And the makefile:

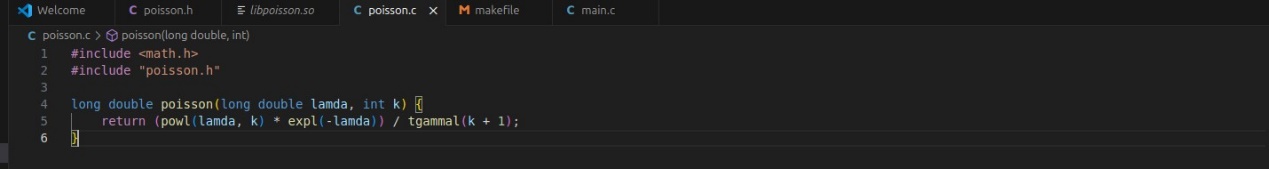


Running example:

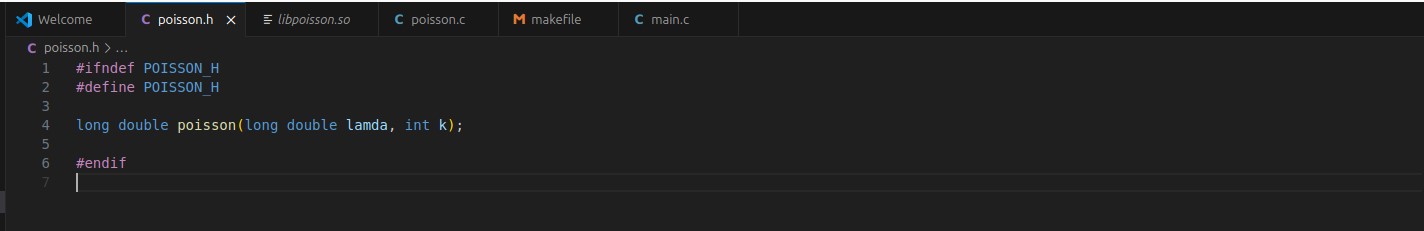


**Exercise 3:**

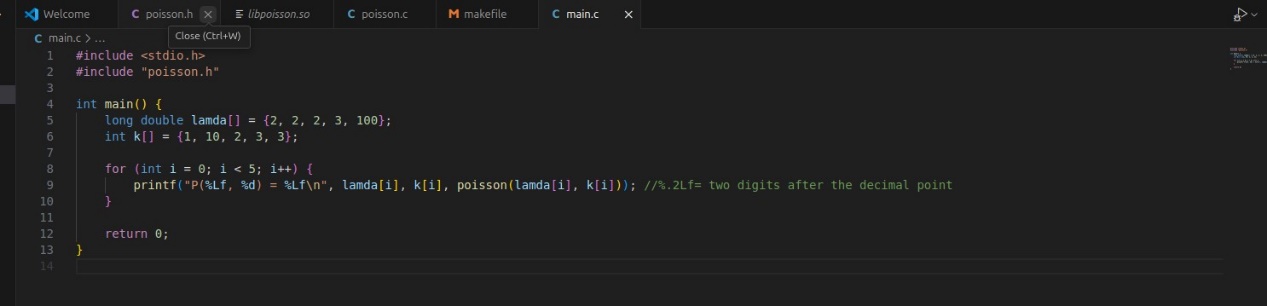
The poisson function as the exercise before:



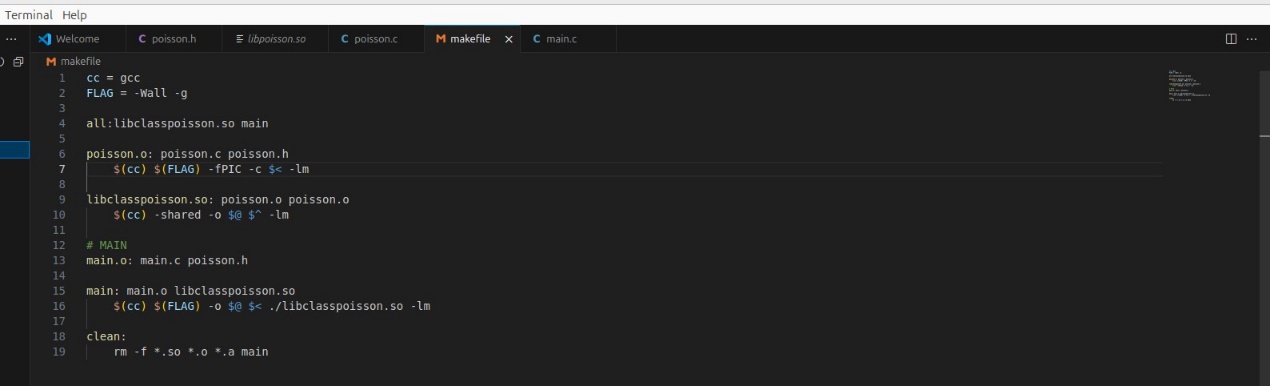
This is the header file:



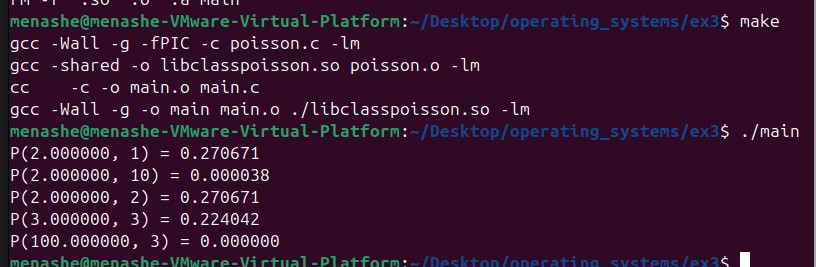
The main function that runs several inputs:



The shared library **libclasspoisson.so** is created from the **poisson.o** object file. This library contains the compiled code from **poisson.c** and is used by the **main** program. The main program includes the header file **poisson.h**, which contains the declarations of the functions defined in **poisson.c**.

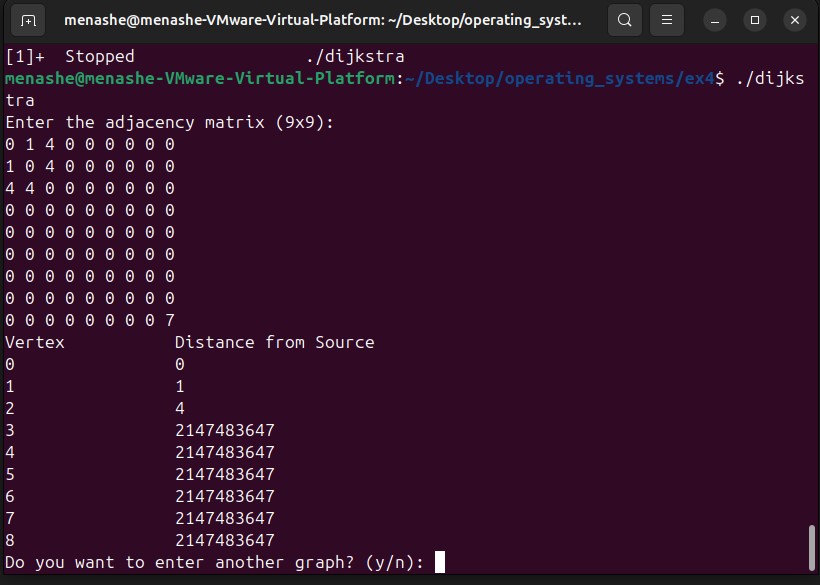


Running example:

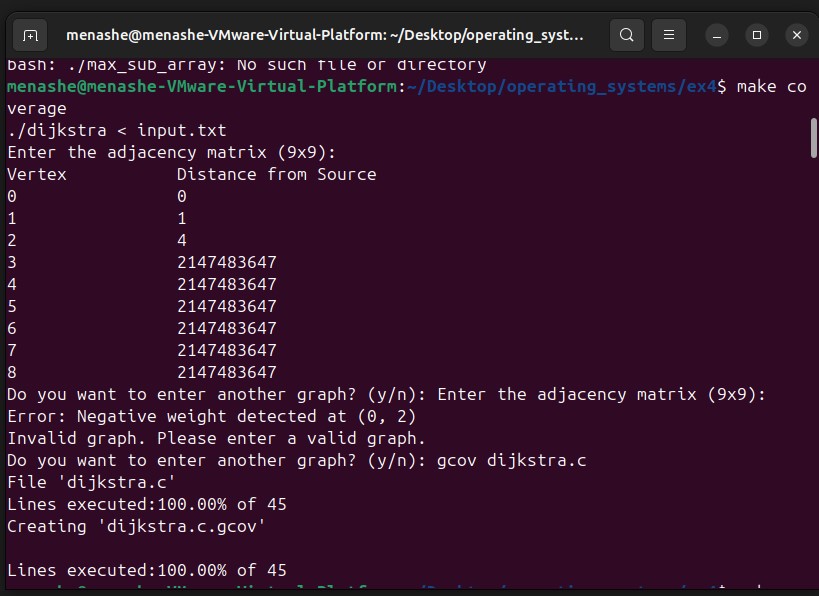


**Exercise 4:**

running example:

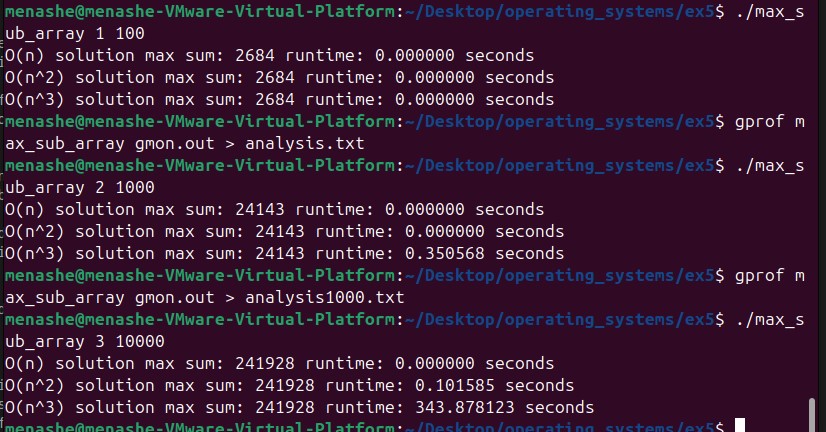


100% with gcov:



**Exercise 5:**

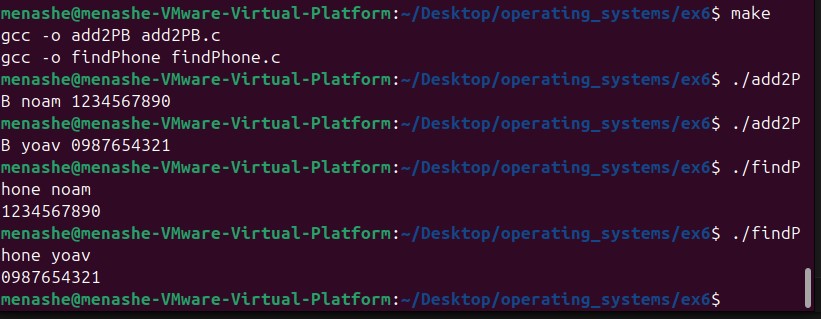
Running example for 100, 1000 and 10000:



Analysis for each one is attached to the code.

**Exercise 6:**

Running example:



Additional explanations are provided within the code itself.

We will use the tar command to compress the assignment - tar -zcvf operating\_systems.tar.gz operating\_systems